

TEMA 6 PROGRAMACIÓ ORIENTADA A OBJECTES : HERÈNCIA

BASE TEORICA

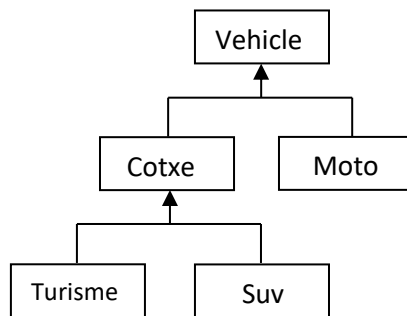
La **herència** és la característica principal de la POO. És una relació entre:

- una superclasse o classe pare: classe més general
- una subclasse o classe filla: classe més especialitzada

Es diu que la subclasse *hereta* les dades (variables) i comportaments (mètodes) de la superclasse. Això ens permet **reutilitzar** el codi:

- Un mètode d'una superclasse és heretat per totes les subclasses. S'escriu un mètode una vegada i es pot fer servir per totes les subclasses.
- Un conjunt d'atributs en una superclasse són heretades per totes les subclasses. Una classe i totes les subclasses comparteixen el mateix conjunt de propietats.
- Una subclasse només necessita implementar les diferències entre ella i el seu antecessor (classe pare).

Per exemple, un cotxe o una moto són vehicles i un turisme o un Suv són cotxes, podem establir la relació:



Veiem que la classe Cotxe i la classe Moto són filles de la classe Vehicle i llavors funcionaran com una extensió o especialització la classe Vehicle, molt més genèrica. De igual manera funcionaran la classe Turisme i Suv que són filles de la classe Cotxe. Per exemple, podem afirmar que:

- “Cotxe” hereta les variables i mètodes de “Vehicle”, “Cotxe” *estén de* “Vehicle”
- “Cotxe” és subclasse de “Vehicle”: és una *classe derivada o filla* de “Vehicle”
- “Vehicle” és *superclasse* de “Cotxe”, llavors és la *classe base o pare* de “Cotxe”
- Les classes “Cotxe” i “Moto” a més a més dels seus propis atributs i mètodes hereten de la classe “Vehicle” tots els seus atributs i mètodes.
- L'herència realitza la relació **és-un**: *Un cotxe és-un vehicle.*

Cal destacar que en Java només és possible la herència simple, es a dir que una classe només pot derivar d'una única classe pare.

- **Implementació de l'herència**

La sintaxi per declarar classes derivades és:

class ClasseDerivada extends ClasseBase { ... }

Implementem la classe Vehicle:

```
public class Vehicle {
    private String marca;
    private int motor;
    private double preu;
    //Constructors
    public Vehicle () {
        marca = " ";
        motor = 0;
        preu = 0.0;
    }
    public Vehicle (String marca, int motor, double preu){
        this.marca = marca;
        this.motor = motor;
        this.preu = preu;
    }
    //Setters (modificadors) i getters (consultors)
    public void setMotor(int motor){
        this.motor = motor;
    }
    public void setMarca(String marca){
        this.marca = marca;
    }
    public void setPreu(double preu) {
        this.preu = preu;
    }
    public int getMotor(){
        return motor;
    }
    public String getMarca(){
        return marca;
    }
    public double getPreu(){
        return preu;
    }
    //Mètode per calcular el preu
    public double calcularPreu() {
        return 0.0;
    }
    //Entrada i sortida
    public void mostrarVehicle(){
        System.out.println("Marca: " + marca + " Motor: " + motor + " Preu: " + preu);
    }
    //Podem redefinir el mètode toString de la classe Object
    @Override
    public String toString(){
        return "Marca: " + marca + " Motor: " + motor + " Preu: " + preu;
    }
}
```

A destacar:

- Hem redefinit o també “re-escrit” (“**override**”) el mètode toString() que pertany a la classe Object que en Java és la classe pare de totes les classes.
- Amb **super()** accedim als membres (atributs i/o mètodes) de la classe pare. En un constructor sempre es posarà en primer lloc la crida al constructor de la classe pare amb *super()*. Si no es posa es cridarà al constructor per defecte de la classe pare. A través de la referència a *super* es pot accedir a qualsevol mètode de la superclasse.

La classe Cotxe és filla de la classe Vehicle:

```
public class Cotxe extends Vehicle {
    private int places;
    private String tipus;
    private final double IMPOST = 1.3;
    //Constructor
    Cotxe(String marca, int motor, double preu, int places, String tipus) {
        super(marca,motor, preu);
        this.places = places;
        this.tipus = tipus;
    }
    //Setters (modificadors) i Getters (consultors)
    public void sePlaces(int places){
        this.places = places;
    }
    public void setTipus(String tipus){
        this.tipus = tipus;
    }
    public int getPlaces(){
        return places;
    }
    public String getTipus(){
        return tipus;
    }
    //Mètode propi
    public final double calcularPreu() {
        return super.getPreu()*IMPOST;
    }
    //Mètodes d'entrada i/o sortida
    public void mostrarCotxe() {
        super.mostrarVehicle();
        System.out.println(" Places: " + places + " Tipus: " + tipus);
    }
    @Override
    public String toString(){
        return super.toString() + " Places: " + places + " Tipus: " + tipus;
    }
}
```

A destacar que:

- Amb la paraula **extends** establim que la classe Cotxe és filla de la classe Vehicle.
- Hem definit un mètode propi per calcular el preu de venda (*calcularPreu()*)
- Atenció, els constructors no se hereten, els de la classe filla han de reescriure els de la classe pare.

- Amb la paraula **final** podem:
 - Establir constants simbòliques (IMPOST)
 - Establir que un mètode no es pugui redefinir en una classe filla
 - Evitar que es puguin crear classes filles d'una classe

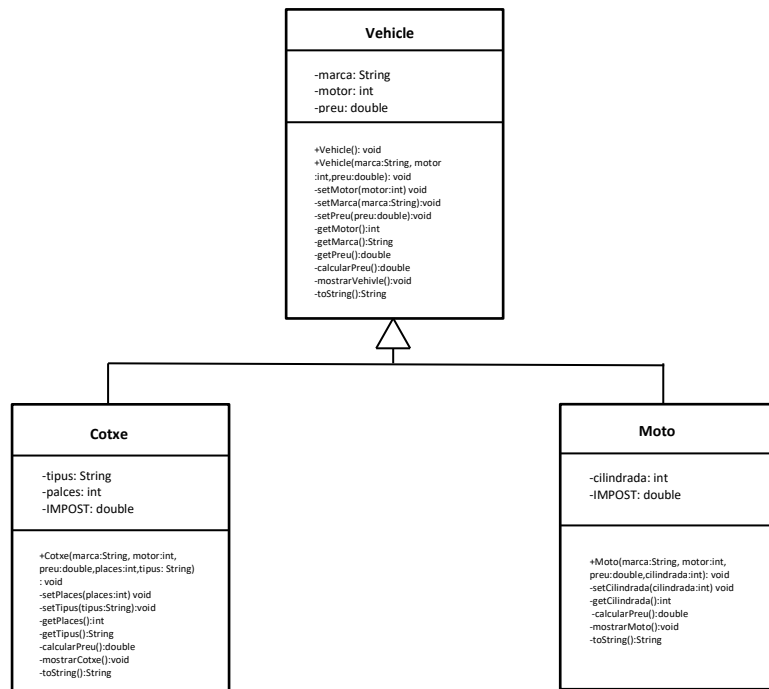
La classe Moto és també filla de la classe Vehicle:

```
public class Moto extends Vehicle {
    private int cilindrada;
    private final double IMPOST = 1.2;
    public Moto(String marca, int motor, double preu, int cilindrada) {
        super(marca,motor,preu);
        this.cilindrada = cilindrada;
    }
    public void seCilindrada(int cilindrada){
        this.cilindrada = cilindrada;
    }
    public int getCilindrada(){
        return cilindrada;
    }
    public double calcularPreu() {
        return super.getPreu()*IMPOST;
    }
    public void mostrarMoto() {
        super.mostrarVehicle();
        System.out.println(" Cilindrada: " + cilindrada);
    }
    @Override
    public String toString(){
        return super.toString() + " Cilindrada: " + cilindrada;
    }
}
```

A destacar que:

- Amb la paraula **extends** establim que la classe Moto és filla de la classe Vehicle.
- Hem definit un mètode per calcular el preu de venda (*calcularPreu()*) que ara **no és final**, llavors es podrà heretar des de qualsevol classe filla de la classe Moto.

El diagrama UML serà:



Ara podem implementar un *main* per provar com podem treballar amb les diferents classes. Podem comprovar com també és possible construir un *vector de Vehicles* i posar dins d'aquest vector els diferents tipus de vehicles.

```
public static void main(String[] args) {
    Vehicle v1 = new Vehicle("Ford", 90, 150000);
    Cotxe c1 = new Cotxe("Seat", 120, 250000,5,"benzina");
    Moto m1 = new Moto("BMW", 90, 10000,250);
    System.out.println(v1);
    System.out.println(c1);
    System.out.println("PREU: " + c1.calcularPreu());
    System.out.println(m1);
    System.out.println("PREU: " + m1.calcularPreu());

    Vehicle Automobils[] = new Vehicle[4];

    Automobils[0] = new Cotxe("SEAT", 120, 250000,5,"benzina");
    Automobils[1] = new Moto("BMW", 90, 10000,250);
    Automobils[2] = new Cotxe("AUDI", 150, 300000,5,"diesel");
    Automobils[3] = new Cotxe("LEXUS", 200, 350000,5,"hibid");

    for(int i = 0; i < Automobils.length; i++) {
        System.out.println(Automobils[i]);
        System.out.println("PREU: " + Automobils[i].calcularPreu());
    }
}
```

- **Sobrecàrrega (“overloading”) dels constructors**

Com ja hem vist en una classe es poden definir mètodes amb el mateix nom sempre i quan els arguments siguin diferents. Aquesta propietat s'anomena “**overloading**” o sobrecàrrega i s'utilitza molt pels constructors. El compilador Java sempre determinarà el mètode que es crida gràcies al nombre de paràmetres que se li passen.

En el nostre exemple, a la classe Vehicle tenim sobrecarregats els dos constructors definits:

```
public Vehicle ()
public Vehicle (String marca, int motor, double preu)
```

- **Redefinint (“overriding”) operacions als constructors i els mètodes**

Una subclasse pot redefinir o re-escriure (“**@override**”, indicació que es recomenable posar al codi) un mètode de la classe pare, de manera que li poden incorporar més funcionalitats o fins i tot fer-la de nou. La redefinició unifica la reutilització i l'extensibilitat del codi.

En ocasions el mètode redefinit des d'una classe filla pot invocar la de la classe pare amb la paraula **super()**, segon el cas:

- dels constructors: *super(paràmetres...);*
- dels mètodes : *super.nomMètode(paràmetres...);*

Els constructors no se hereten, les subclasses han de definir el seu propi constructor i sempre el primer que han de fer és cridar al constructor de la classe pare amb **super()**, si no es fa explícitament el compilador de Java cridarà al constructor per defecte de la

superclasse i si no el té hi haurà un error de compilació. Per una altra banda, si la classe té algú constructor definit, el constructor per defecte no existirà, llavors si volem que hi hagi un constructor sense paràmetres l'hauré de declarar explícitament.

Cal tenir clar que:

- **Re-escritura:** La subclasse substitueix la implementació d'un mètode de la superclasse. Els dos mètodes han de tenir la mateixa signatura o capçalera.
- **Sobrecàrrega:** Existeix més d'un mètode amb el mateix nom però amb signatura diferent (amb un nombre de paràmetres diferent). Els mètodes sobrecarregats poden definir-se en la mateixa classe o en classes diferents de la jerarquia de herència.

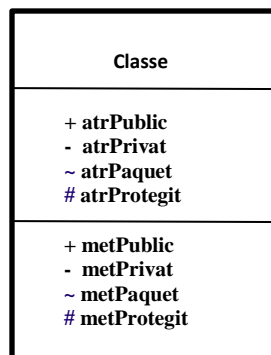
Cal tenir en compte que si una classe filla anomena un atribut amb el mateix nom que un atribut de la classe pare, aquest últim no serà accessible. Es diu llavors que el nou atribut de la classe filla “*amaga*” l'atribut de la classe pare.

- **El modificador protected**

Els modificadors d'accés (“**encapsulació**”, es a dir ocultació) tant per mètodes com atributs són:

- **<cap>**: accessible des de el paquet
- **public**: accessible des de tot el programa
- **private**: accessible només des de la pròpia classe
- **protected**: accessible des de el paquet i des de les subclasses

En UML:




Normalment no és una bona pràctica fer-ne ús de *protected*, i només s'utilitza per casos concrets d'atributs on interessa que pugui ser modificat des de qualsevol subclasse.

- **El modificador static**

Els mètodes i/o atributs declarats com a estàtics (“**static**”) només es creen un per classe i no un per cada objecte que s'hagi creat amb aquesta classe, amb la qual cosa representen un gran estalvi en memòria ja que només ocuparan un espai en memòria i no un per cada instància. A més a més seran compartits per tots els objectes creats de la classe. Com que és propi de la classe, un membre estàtic (atribut i/o mètode) s'utilitza directament amb el nom de la classe.

Per exemple, per:

```
public class A {  
    private int num;  
    //Atribut estàtic  
    private static String nom = "one";  
    ...  
}
```



```
System.out.println ("Nom: " + A.nom);
```

Un mètode estàtic no pot accedir a membres no estàtics directament, s'haurà de crear primer un objecte de la classe.

- **El modificador final**

El modificador final es pot aplicar en diferents casos, ens indica segons el context que no es pot fer cap canvi:

- Paràmetres: Indica que dins del mètode no podrem modificar el valor d'aquest paràmetre:

```
public void miMetodo(final int p1, int p2){}  
//no podrem modificar el valor p1
```

- Atributs: Indica que dins de la classe no podrem modificar el valor d'aquest atribut. S'utilitza per a definir constants juntament amb static:

```
public static final double PI = 3.14;  
//no podrem modificar el valor assignat a PI
```

- Mètodes: Indica que les classes que heretin d'aquestes no podran sobre escriure aquest mètode.

```
public final void myMethod(){}  
//no podrem sobre escriure myMethod
```

- Classes: No es pot estendre la classe. No es pot "heretar d'ella".

```
public final class myClass(){}  
//no podrem fer: extend myClass
```

- **Polimorfisme**

Aquest concepte ("múltiples formes") s'aplica a la propietat que proporciona Java de que un mateix mètode es pot aplicar a objectes diferents, concretament això es tradueix en dos propietats:

- Una referència a una superclasse pot referir-se a un objecte de qualsevol de les seves subclasses:

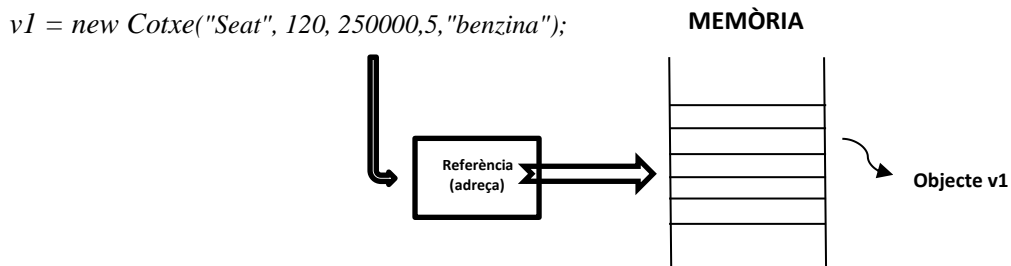
```
Vehicle v1 = new Cotxe("Seat", 120, 250000,5,"benzina");
```

- El mètode es selecciona en base a la classe de l'objecte utilitzat, i no al de la referència. D'aquesta manera, si utilitzem el mètode toString() o calcularPreu() sobre v1, s'utilitzaran els de la classe Cotxe i no els de la classe Vehicle.

Cal tenir clar la diferencia entre referència i objecte. Quan fem:



la declaració de la variable `v1` amb el tipus de la classe `Vehicle` (objecte) no implica la creació de l'objecte, només el prepara per contenir la referència (lloc de memòria, que de moment és desconegut i per tant **null**) on posarà l'objecte un cop creat. Per crear l'objecte hem d'utilitzar l'operador **new**, amb el qual assignem un lloc a la memòria on es col·locarà l'objecte creat:



En conclusió, Java permet que una referència a una superclasse pugui apuntar a un objecte de qualsevol de les seves subclasses, dit d'una altra manera, una crida a un mètode sobre una referència d'un tipus general (classe base) executarà la implementació corresponent del mètode de la classe del objecte que realment s'hagi creat (classe derivada), es a dir Java té la capacitat de buscar en cada moment el mètode que ha d'executar en funció del tipus d'objecte dins d'una jerarquia d'herència establerta. Aquesta propietat se li diu “**enllaç dinàmic**”, ja que aquest procés es fa en temps d'execució del programa i no en el procés de compilació.

Gràcies al polimorfisme es possible que un array contingui referències a objectes de classes diferents: la superclasse i totes les seves subclasses:

```
Vehicle Automobils[] = new Vehicle[4];

Automobils[0] = new Cotxe("SEAT", 120, 250000,5,"benzina");
Automobils[1] = new Moto("BMW", 90, 10000,250);
Automobils[2] = new Cotxe("AUDI", 150, 300000,5,"diesel");
Automobils[3] = new Cotxe("LEXUS", 200, 350000,5,"hibid");

for(int i = 0; i < Automobils.length; i++) {
    System.out.println(Automobils[i]);
    System.out.println("PREU: " + Automobils[i].calcularPreu());
}
```

Noteu com a més a més, el mètode `calcularPreu()`, gràcies també al polimorfisme, s'adapta al tipus d'objecte que hi ha al vector `Automobils` en cada cas.

- **Casting (Upcasting i Downcasting)**

Podem convertir referències entre classes i subclasses (només) gràcies al casting:

```
Vehicle v
Cotxe c = new Cotxe(...);
v = c           //càsting implícit tipus upcasting, tot Cotxe és un Vehicle
Cotxe c1 = (Cotxe)v; //càsting explícit tipus downcasting
```


Amb el *casting* canvia com veiem i tractem a un objecte:

- a través de *v* es veu com un *Vehicle* i podrem invocar mètodes d'aquesta classe.
- a través de *c1* es veu com un *Cotxe* i podrem invocar mètodes d'aquesta classe.

Un objecte de la classe derivada sempre es podrà utilitzar en lloc d'un objecte de la classe base (ja que es compleix la relació "és-un"). En aquest cas s'anomena **upcasting**, la conversió és cap a dalt. El cas contrari s'anomena **downcasting**, i no sempre funciona ja un objecte de la classe base no sempre és un objecte de la classe derivada.

Java proporciona l'operador **instanceof** que comprova si la classe d'un objecte, és realment d'una determinada classe:

Objecte instanceof Classe

Per exemple: *v instanceof Cotxe*

retorna true si *v* apunta a un objecte de la classe *Cotxe* o de qualsevol de les seves subclasses.

- **La classe Object**

En Java totes les classes deriven de la classe *Object* (paquet: *java.lang.Object*). El compilador per defecte afegeix "**extends Object**" a totes les classes:

public class Classe {...} → el compilador fa → *public class Classe extends Object {...}*

Gràcies a que qualsevol mètode d'una superclasse es pot redefinir en les seves subclasses, qualsevol mètode de la classe *Object* es pot redefinir a les nostres classes. Per exemple:

- **el mètode equals**: compara referències a objectes i està definit a la classe *Object*:

```
public class Object {  
    ...  
    public boolean equals(Object obj) {  
        return this == obj;  
    }  
    ...  
}
```

A la nostra classe *Cotxe* podem incorporar la redefinició d'*equals*:

```
@Override  
public boolean equals(Object obj) {  
    if (!(obj instanceof Cotxe)) return false;  
    Cotxe c = (Cotxe) obj;  
    return places == c.places && tipus.equals(c.tipus);  
}
```

I podrem fer: *if(c1.equals(c2))....*

- **el mètode toString**: concatena un objecte amb un *String* i també està definit a la classe *Object*:

```

public class Object {
    ...
    public String toString( ) {
        return ...;
    }
    ...
}

```

Es utilitzat quan es concatena un objecte amb un String. Per exemple:

```
System.out.println(v1);
```

per defecte retorna un String amb el nom de la classe i l'adreça de memòria de l'objecte: Vehicle@a34f5bd

A la nostra classe Vehicle podem incorporar la redefinició de toString:

```

@Override
public String toString() {
    return "Marca: " + marca + " Motor: " + motor + " Preu: " + preu;
}

```

I podrem fer: `System.out.println(v1);`

provocant la sortida per pantalla: `Marca:Ford Motor:90 Preu:150000`

Altres mètodes de la classe Object:

- `public Object clone()`: clonació d'objectes
- `public Class getClass()`: classe a partir de la que ha sigut instanciat un objecte.
- `public int hashCode()`: codi hash utilitzat en les col·leccions (es veurà en el següent tema).

BASE PRÀCTICA

- **Programes per desenvolupar**

1.- Afegiu les classes Turisme i Suv derivades de la classe. Implementeu-les amb els constructors, getters i setters i altres mètodes d'entrada i sortida que creieu convenient. Doneu el codi de les dues classes i el diagrama UML complet de les cinc classes.

2. – Feu un *main* on en un vector de Vehicles anomenat Automòbils es col·loquin 10 vehicles del tipus: 8 Cotxes (4 Turismes i 4 Suv) i 2 Motos. Com a sortida s'ha de fer un llistat de tots els vehicles que es disposa amb les seves característiques i el preu de venda.

3.- Feu les comprovacions pertinents amb l'aplicació concessionari muntada fins ara i contesteu a les següents qüestions de manera raonada:

a) Comproveu l'efecte de definir els atributs *places* i *tipus* de la classe Cotxe com a protected. Què resulta més adequat?

b) A la classe Cotxe tenim: *public final double calcularPreu()*
Comproveu i detal·leu l'efecte de tenir *final* o no a la definició d'aquest mètode. Resulta convenient aquesta definició en vistes del que necessitem a la nostra aplicació?

c) Redefiniu el mètode *calcularPreu()* per les dues classes noves: Turisme i Suv. Comproveu el seu funcionament per un IMPOST definit per cadascuna de 1.25 i 1.4 respectivament.

d) Tant a la classe Cotxe com a la classe Moto s'ha definit:

```
private final double IMPOST
```

Resulta més convenient posar: *private final static double IMPOST*
Quin és el motiu i feu una comprovació demostrativa.

e) Volem comparar i determinar si dos turismes són iguals, redefiniu el mètode *equals* a la classe Turisme i comproveu el seu efecte.

f) En virtut de quina propietat podem fer:

```
Vehicle v2 = new Cotxe("Audi", 180, 250000, 5, "diesel");
```

Poseu al vostre *main* el tros de codi:

```
Vehicle v2 = new Cotxe("Audi", 180, 250000, 5, "diesel");  
Cotxe c = v2;  
System.out.println(v2);  
System.out.println("PREU: " + v2.calcularPreu());  
System.out.println(c);  
System.out.println("PREU: " + c.calcularPreu());
```

Comproveu que hi ha un error, expliqueu quin és l'error i a què es deu i com es pot solucionar.

f) Gràcies al polimorfisme podem aplicar un mateix mètode a un objecte i Java és capaç de seleccionar el mètode segons l'objecte i la referència. Poseu algun exemple on es posi de rellevància aquesta propietat en el vostre programa.

g) Dibuixeu el diagrama UML de l'aplicació incorporant la classe Object i sense detallar els atributs ni els mètodes de les classes.



By: *Alfons Valverde Ruiz*